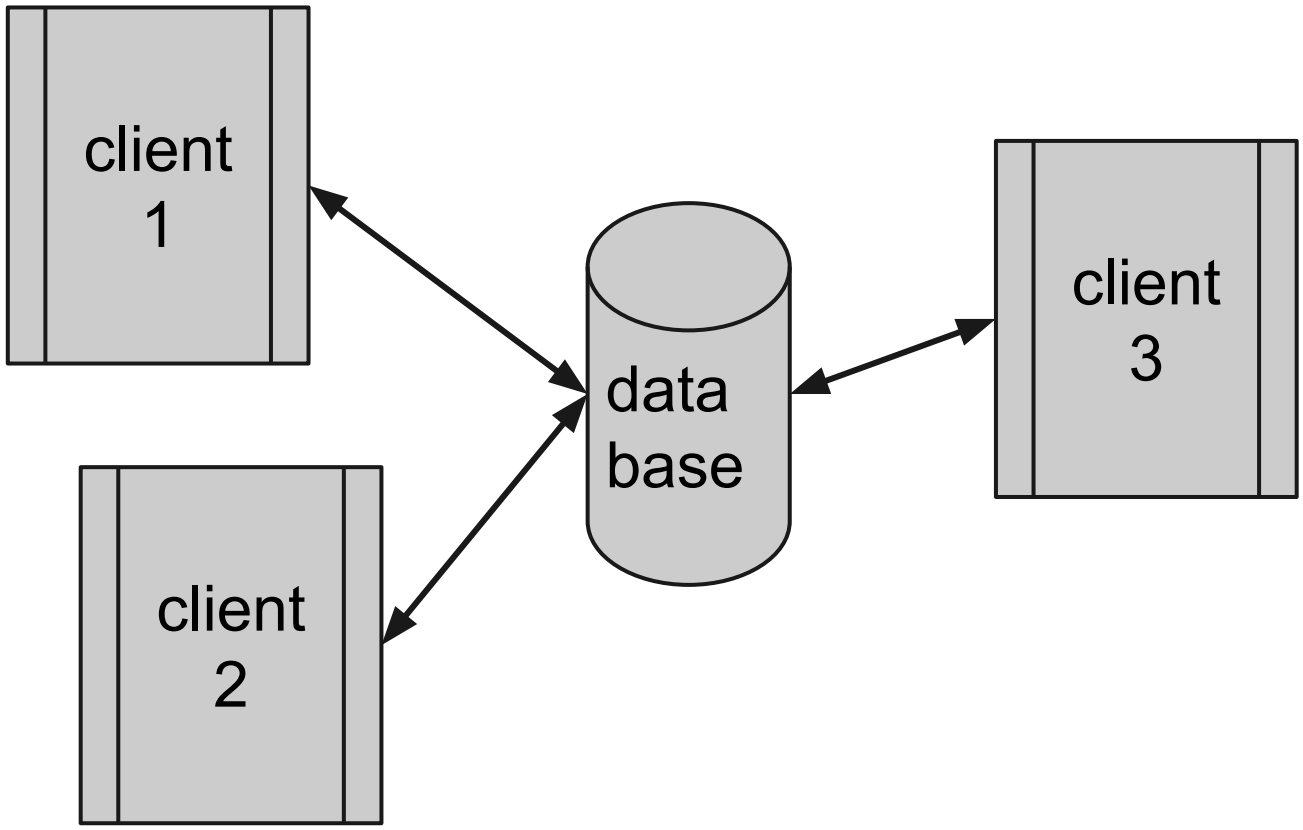
Concurrency in databases

intro to implementation

Outline

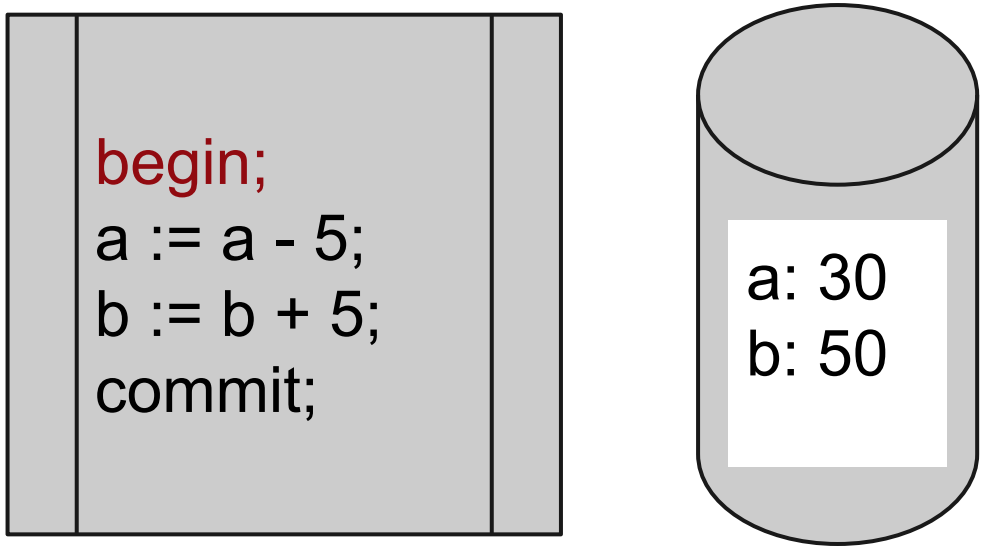
- transactions and concurrent access
- isolation levels
- implementation
  - locking
  - optimistic (MVCC)

What is concurrency?



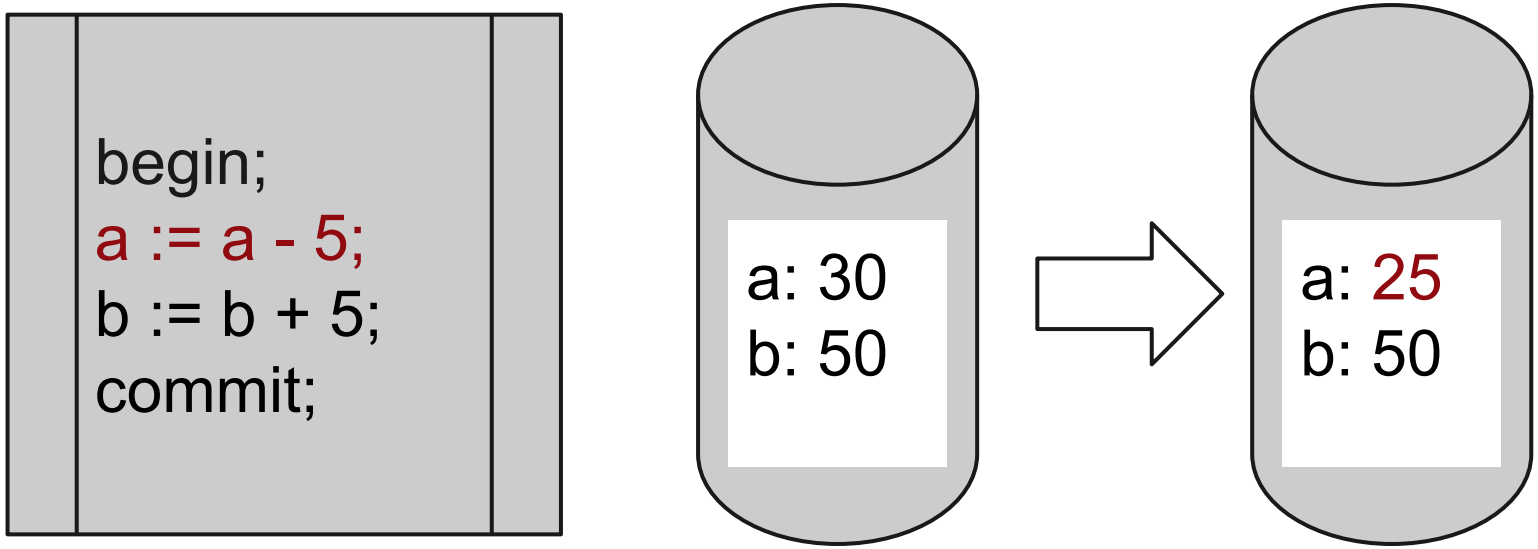
What is a transaction?

Note that a+b=80. This is a constraint.



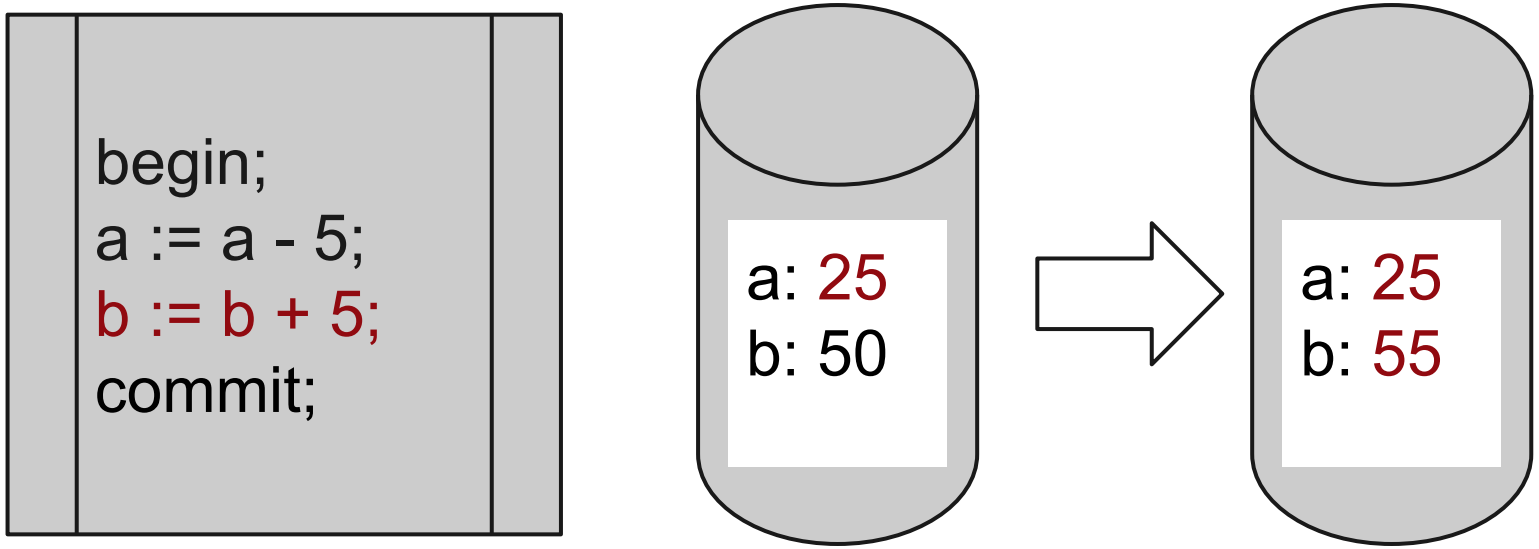
What is a transaction?

The transaction has started, a+b=75



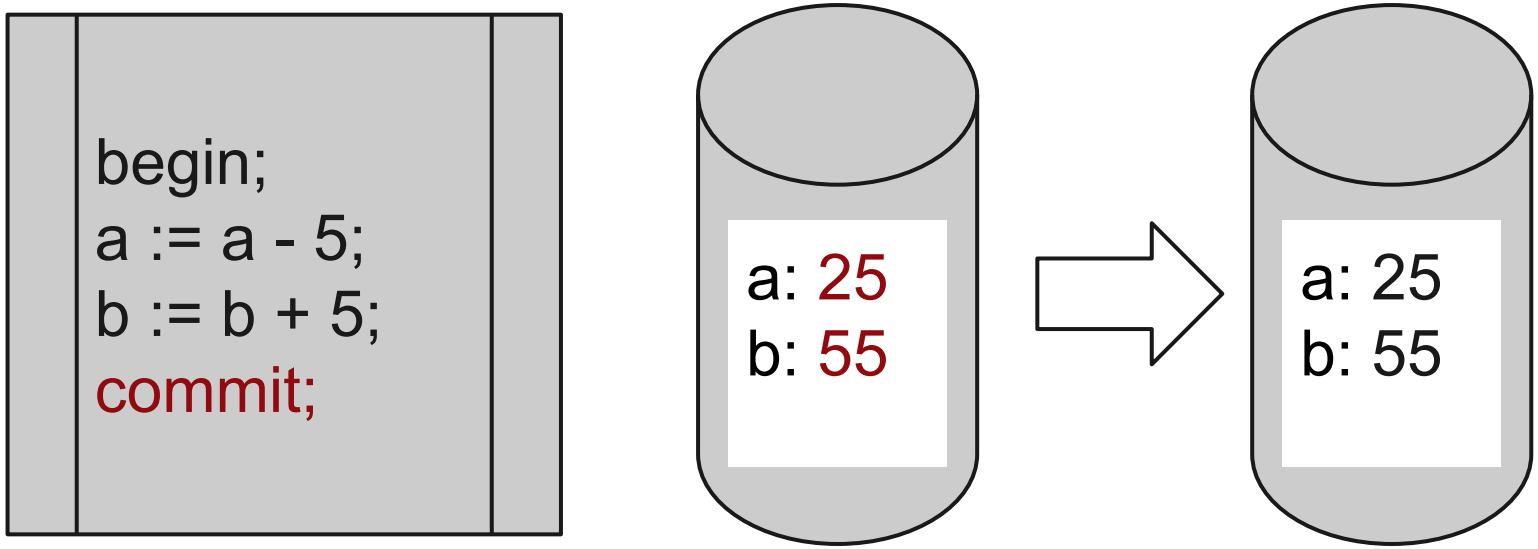
What is a transaction?

Now the constraint is satisfied again, but the changes haven't finalized



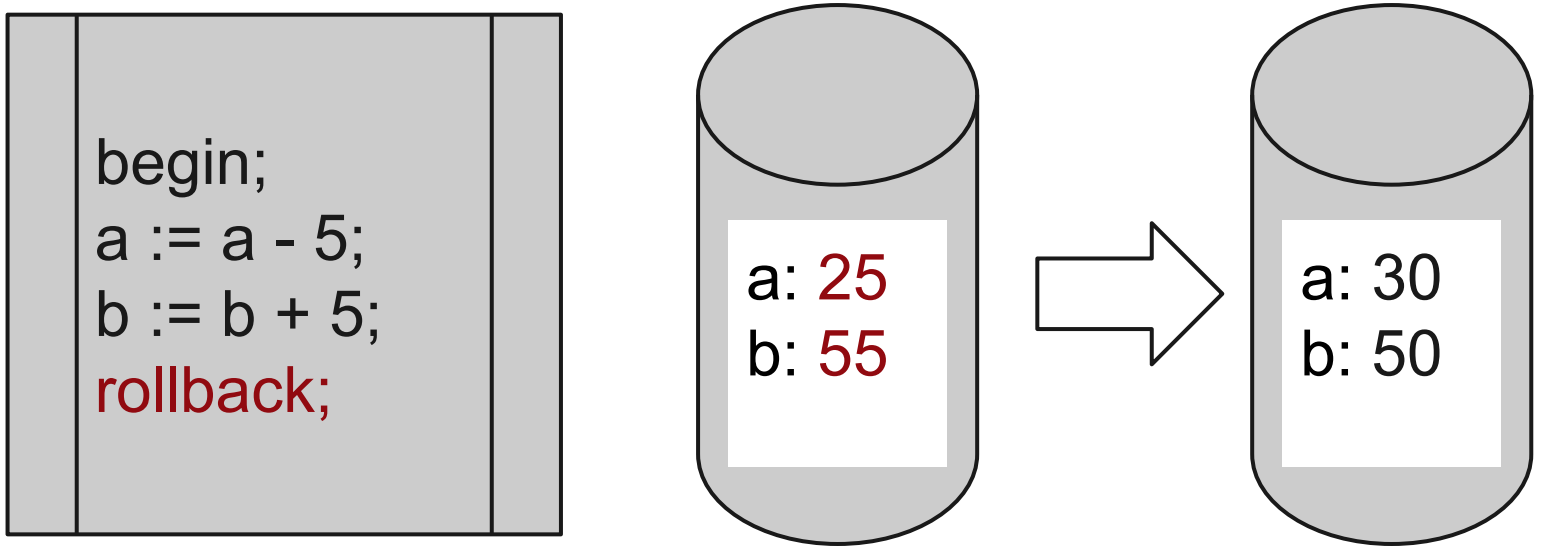
What is a transaction?

COMMIT makes the changes permanent



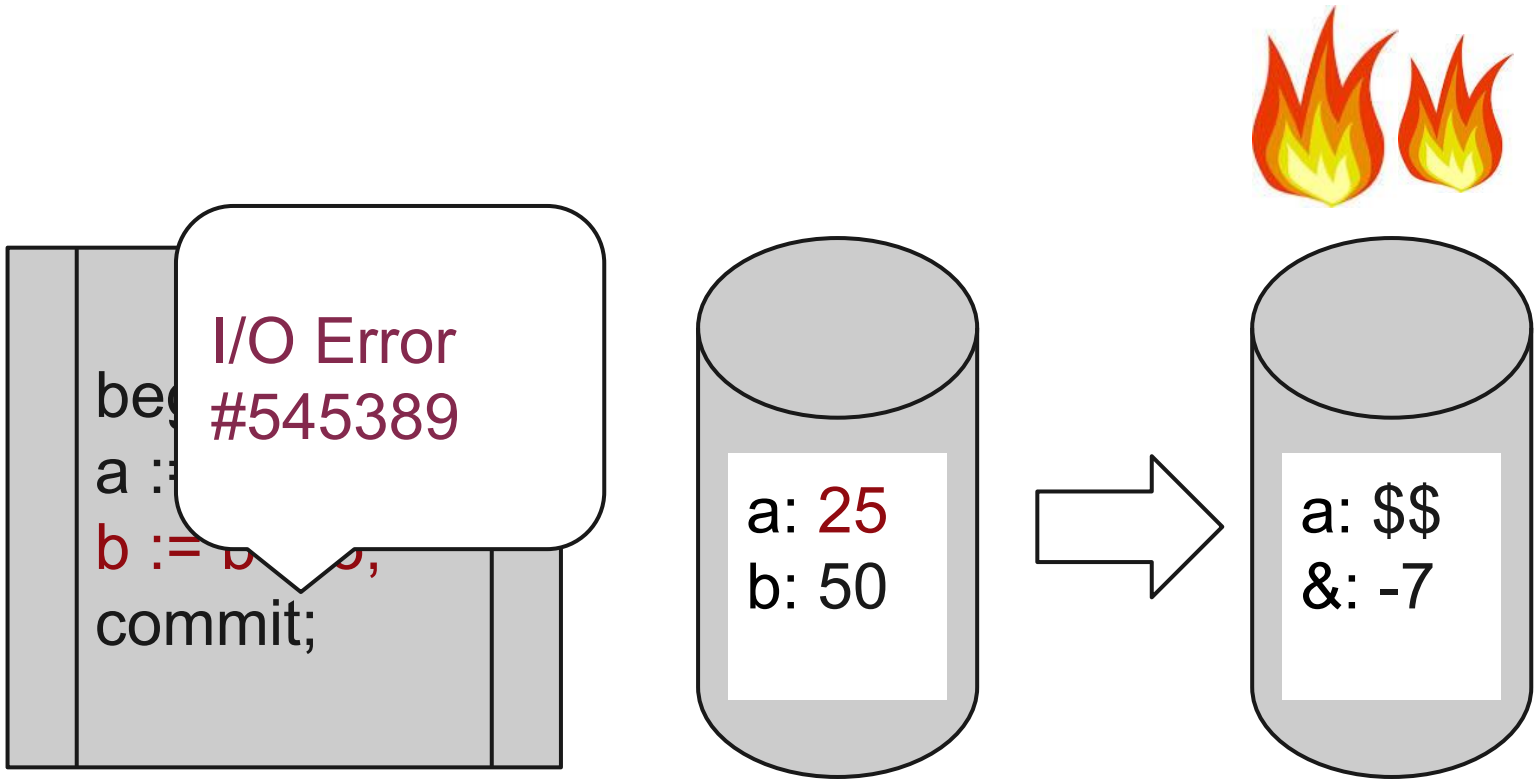
What is a transaction?

ROLLBACK makes it so as if nothing had happened.



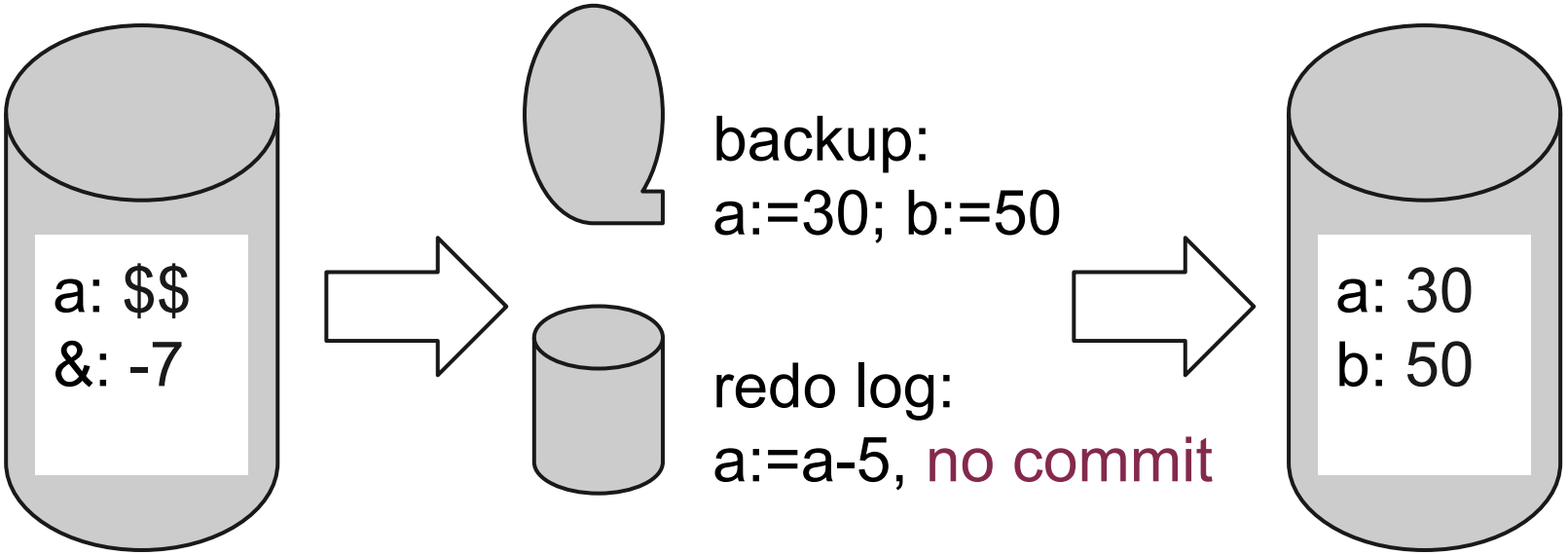
Transaction atomicity

Errors happen



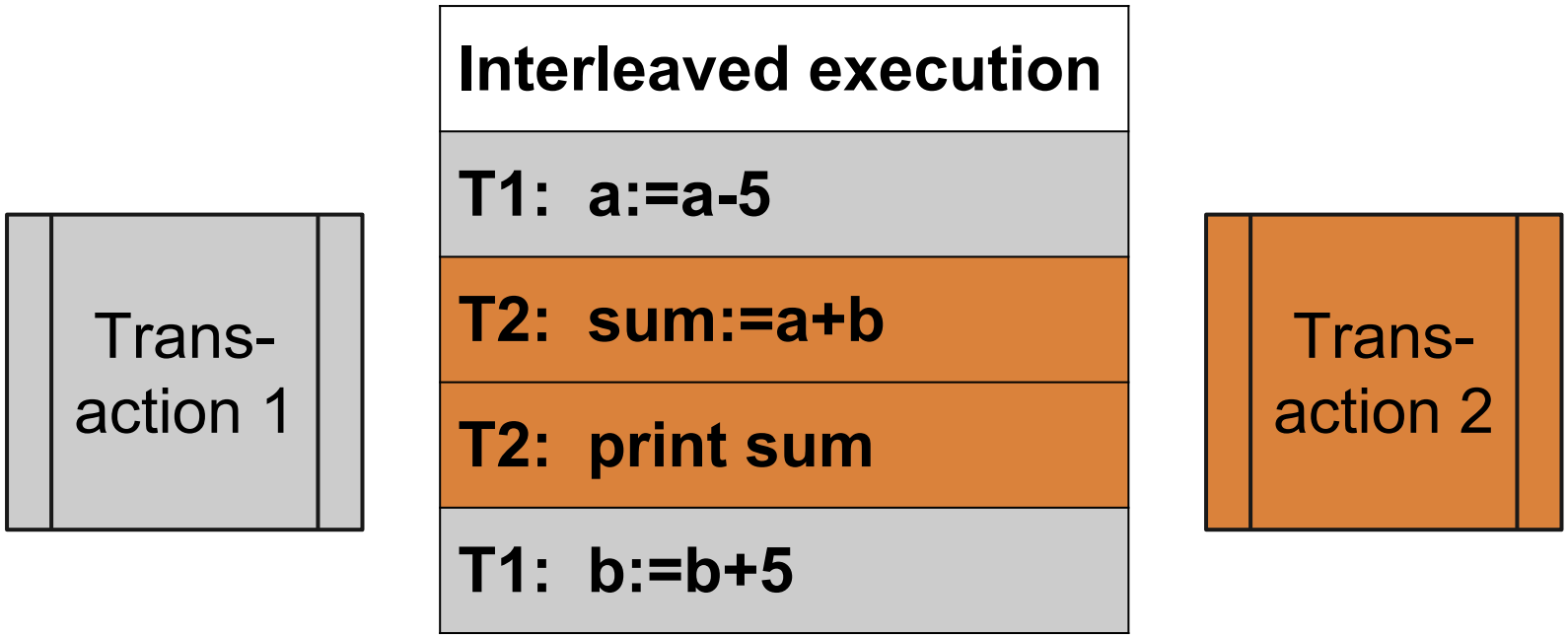
Transaction atomicity

The recovery procedure does not restore incomplete transactions



Transaction concurrency

In multiprocessing environments, processes sleep and wake up frequently



Database isolation

SQL-92 standard: how the database should behave in situations like this. There are 4 levels:

- READ UNCOMMITTED - transactions see data that other transactions are writing

Database isolation

- READ COMMITTED - transactions only see committed data

- REPEATABLE READ - if a transaction repeats a query, the data that was returned in the first query does not change in the second query (even if some other transaction commits)

Database isolation

- SERIALIZABLE - the database behaves exactly like the transactions would have executed one by one (serially), not in parallel.

The default level in Postgres is “READ COMMITTED”. (Suprise?)

Useful notation

How to quickly note down operations of interleaved transactions (histories):

r1[x] - T1 reads data item x

w2[x] - T2 writes data item x

a2, c1 - T2 aborts (rolls back), T1 commits; in this order

Concurrency control

Two main approaches:

- locking: block any operation that would result in an anomaly.
- optimistic: allow everyone read and write. Hope that nothing bad happens, but if it does, fix the results.

Concurrency control

Try something simple first:

Completely serialize access to database

l1, ...T1… *u1, l2,* … T2 … *u2* etc.

l locks the database, *u* unlocks it.

Concurrency control

Slightly better is to allow read-only transactions to go in parallel:

sl1, r1[x], sl2, r2[y], u2, u1, xl3, w3[z], u3.

(T3 waits until T1,T2 finish)

sl - get a shared (read) lock

xl - get an exclusive (write) lock

Concurrency control

Look at this history again:

sl1, r1[x], sl2, r2[y], u2, u1, xl3, w3[z], u3.

Each transaction accesses a different data item. T3 shouldn’t have to wait until T1 and T2 complete.

Locking protocol

- Each lock is at a data item level (*xl1[x]* - T1 gets an exclusive lock on x)
- Hold X (exclusive) locks until end of transaction to prevent dirty reads
- Hold S (shared) locks until end of transaction to prevent unrepeatable reads

Strict 2PL

This protocol is called Strict Two-Phase Locking. It is equivalent to:

- SERIALIZABLE, if locks can cover ranges like “WHERE AGE>23”
- READ COMMITTED, if we relax the protocol and release S locks after read operations

Strict 2PL

We can also support the FOR UPDATE clause, if we add a third lock mode, the U (update) lock.

Lock mode compatibility:

	S	**X**	**U**
S	Y	N	N
X	N	N	N
U	Y	N	N

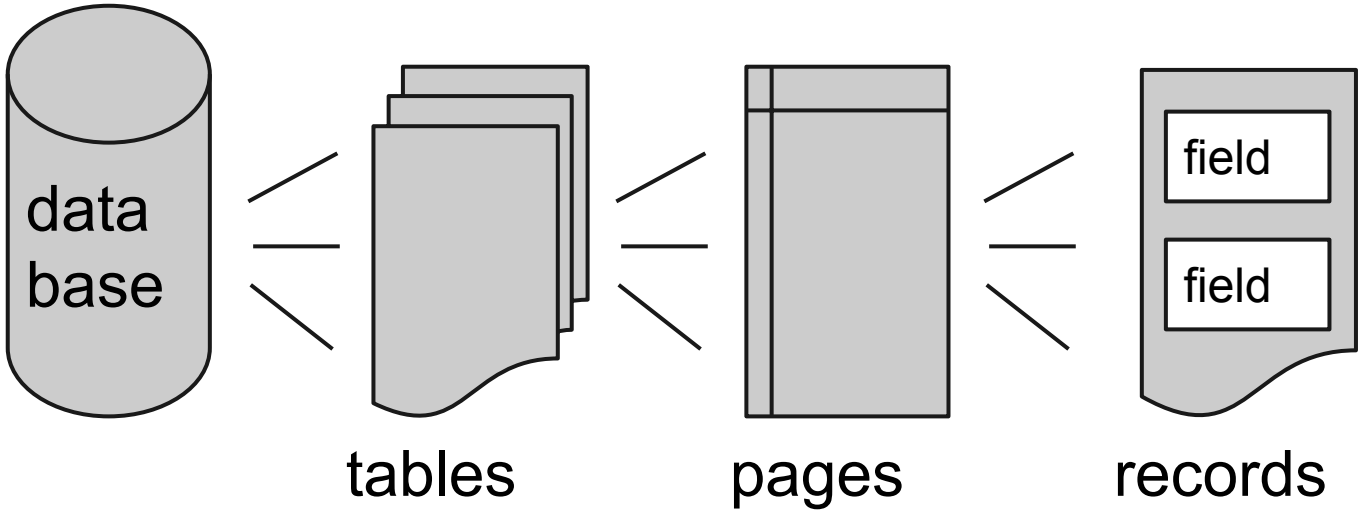
Strict 2PL

We get the isolation level called “Cursor stability” if:

- U locks are taken when the “FOR UPDATE” clause is present
- U locks are converted to X locks
- U and X locks held until end of transaction.

Multiple granularity

Size of “data item” hasn’t been specified yet (you might have guessed “row”).



Multiple granularity

Choice of granularity matters:

- fine granularity (row level): more parallel but more work and space needed to maintain locks
- coarse granularity: less work and space required but less parallel

Multiple granularity

Idea: try to compromise by locking at row level, but switching to larger units if many rows need to be locked.

- if a table is locked, row locks have to be compatible with the table lock
- if some rows are locked, table lock has to be compatible with all of them

Multiple granularity

"Intent" mode locks tell us that lower level locks exist.

- IX - intent to get X lock at lower level
- IS - intent to get S lock at lower level

Compatibility:

	S	**X**	**IS**	**IX**
S	Y	N	Y	N
X	N	N	N	N
IS	Y	N	Y	Y
IX	N	N	Y	Y

Multiple granularity

The rules:

- To get an X or IX lock, need IX lock at parent level.
- To get an S or IS lock, need IS lock at parent level.

Getting each lock starts at the root (database or table level).

Optimistic concurrency control

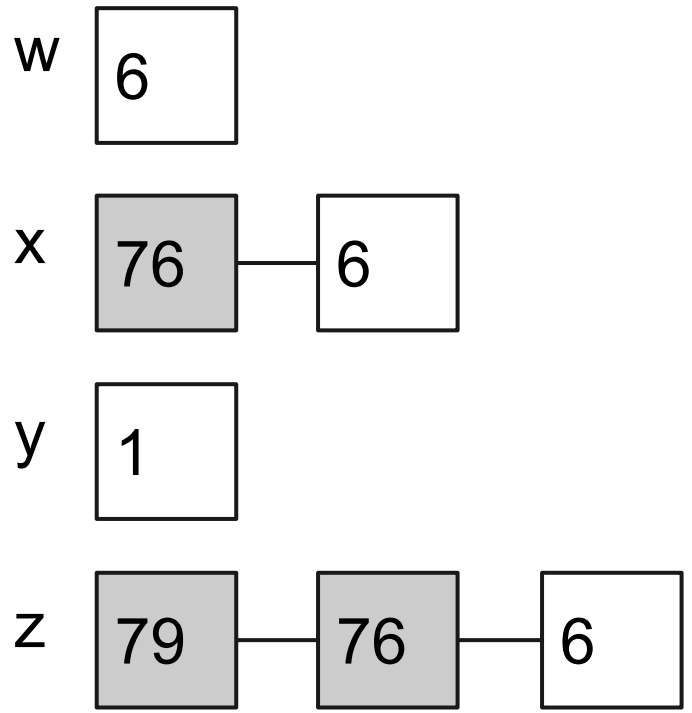
We will look at Multiversion Concurrency Control (MVCC)

Other methods include:

- timestamps
- validation

Pure MVCC

Each change creates a new copy of a data item. The state of database at a given time ("snapshot") is preserved by version timestamps.



State at 5<TS<76

Pure MVCC

Each data item version x_i has:

$RT(x_i)$ - timestamp of last read

$WT(x_i)$ - timestamp of creation (write)

$C(x_i)$ - commit bit

Each transaction receives a timestamp TS(T) at the start.

Pure MVCC: read

```
find the newest x_i such that WT(x_i)<=TS(T)
if C(x_i) /* committed */
  read x_i;
  RT(x_i) := max(TS(T), RT(x_i))
else
  wait until C(x_i)
    or T (TS(T)==WT(x_i)) aborts
```

Pure MVCC: write

```
find the newest x_i such that WT(x_i)<=TS(T)
if TS(T) >= RT(x_i)
  if WT(x_i) == TS(T)
    write x_i; /* overwrite my older write */
  else
    write x_j; /* new version */
    WT(x_j) := TS(T); unset C(x_j)
else  /* can't change the past */
  restart with new TS(T)
```

Pure MVCC

- COMMIT sets $C(x_i)$ bit on all written items, transactions waiting for the $C(x_i)$ bit can continue
- ROLLBACK unblocks transactions waiting for the $C(x_i)$ bit, they will restart their read operation

These operations could be expensive.

Pure MVCC

History purge: find newest x_i such that $WT(x_i)$ is smaller than TS(T) of oldest active transaction.

Then delete all x_j ($j<i$)

MVCC + 2PL

Mainstream databases (like Oracle, Postgres) use a mixed method instead:

- Transactions are classified as “read-only” and “updaters”
- handle read-only through MVCC
- handle updaters through Strict 2PL

MVCC + 2PL

Data item versions have less metadata:

just $TS(x_i)$ - the creation time. There is also a global TS_{curr} which gets assigned to new transactions.

MVCC + 2PL: reader

Read-only transactions only use one operation: “read”.

find the newest x_i such that TS(x_i)<=TS(T);
read x_i

Commit and rollback can be no-ops.

MVCC + 2PL: updater

Updaters read the same way, but they also need the write operation:

```
get exclusive lock on x;
create a new version x_j;
TS(x_j) := +infinity; /* invisible to others */
```

Note that the X lock is not released.

MVCC + 2PL: updater

Commit operation is the only tricky part

```
block other commits;
for all x_i that the transaction created
  if exists x_j (TS(T)<TS(x_j)<TS_curr+1)
    abort with a write-conflict error
  TS(x_i) := TS_curr+1
TS_curr := TS_curr+1; /* new values now visible */
unblock other commits;
release all X locks held
```

MVCC + 2PL: updater

"First committer wins" rule

```
block other commits;
for all x_i that the transaction created
  if exists x_j (TS(T)<TS(x_j)<TS_curr+1)
    abort with a write-conflict error
  TS(x_i) := TS_curr+1
TS_curr := TS_curr+1; /* new values now visible */
unblock other commits;
release all X locks held
```

MVCC + 2PL: updater

- Rollback simply needs to delete all the versions that the transaction created. Others will never see them.
- History purge works like in pure MVCC

Updater transactions need to keep a list of items they've written to.

MVCC + 2PL

This algorithm gives us the SNAPSHOT isolation level (not part of SQL language, but widely acknowledged).

We can relax it by removing the “first committer wins” rule, if each statement also gets new TS (T) := TS_{curr}, the isolation level is READ COMMITTED.

Who uses what

- Strict 2PL: old databases, IBM DB2, MS SQL Server
- MVCC+2PL: Oracle, Postgres, MySQL (InnoDB), MS SQL Server, Apache HBase

NoSQL bases frequently have partial or no support for transactions

Final remarks

We didn’t cover:

- anomaly types
- deadlocks
- concurrent index access
- recovery (how to roll back)

Concurrency-related anomalies can occur in other settings too, not just RDBMS-s.

Useful books

H. Garcia-Molina, J. Ullman, J. Widom
"*Database Systems: The Complete Book*"

A. Silberschatz, H. Korth, S. Sudarshan
"*Database System Concepts*"

P. Bernstein, V. Hadzilacos, N. Goodman
"*Concurrency control and recovery in database systems*"